Automatically Evolving Lookup Tables for Function Approximation

Oliver Krauss[1,2][0000−0002−8136−2606] and
William B. Langdon[3][0000−0002−6388−4160]

[1] Johannes Kepler University Linz, Austria
[2] AIST, University of Applied Sciences Upper Austria
`oliver.krauss@fh-hagenberg.at`
[3] CREST, Computer Science, UCL, UK `W.Langdon@cs.ucl.ac.uk`

Abstract. Many functions, such as square root, are approximated and sped up with lookup tables containing pre-calculated values.
We introduce an approach using genetic algorithms to evolve such lookup tables for any smooth function. It provides double precision and calculates most values to the closest bit, and outperforms reference implementations in most cases with competitive run-time performance.

Keywords: Genetic Improvement · Objective Function · Covariance Matrix Adaptation

1 Introduction

Newton-Raphson [11] is a widely applied method to approximate smooth mathematical functions. We present an approach that allows the fully automated generation of a lookup table for any given mathematical function, across a defined range. Newton-Raphson requires a known approximation and its derivative. Our approach is more accurate than comparable approximation methods and, except where hardware acceleration is provided, e.g. square root, it is also faster.

To validate the method we compare our approach to several reference implementations, including square root and cube root, in C, C++, Java. We give a detailed overview of the design of the fitness function and influencing factors, such as algorithm design and the occurrence of inflection points in the functions to be approximated.

The approach improves the performance of Newton-Raphson by reducing the amount of iterations required, and provides:

- High precision function approximation - all functions are calculated with double precision accuracy, and are more accurate than reference implementations in most cases.
- Auto generated lookup tables - Lookup tables are automatically generated without the need for configuration.
- Fast run-time performance compared to algorithms that are not hardware accelerated.

The approach can be directly applied to various domains. Gauss-Newton, a modification of the original Newton-Raphson algorithm, is used in Genetic Programming [6] to guide the search through the search space [14]. Lookup tables are also used in Genetic Programming as function lookups [2,9]. Newton-Raphson is used in distributed optimization to drive the consensus between the different agents on a shared optimization problem, both synchronously [12] and asynchronously [1]. It is also used in Data Science to solve equations to categorize, group or predict data. In the case of Yap et al. [13], it is used for parameter estimation to fit the Lee-Carter numerical forecasting algorithm. When hardware architectures do not offer hardware acceleration, the square root and division operations as defined in the IEEE Standard for Floating-Point Arithmetic [10,5] are often implemented using Newton-Raphson in reference implementations. To further speed up these implementations reference implementations provide lookup tables to reduce the amount of Newton-Raphson iterations needed.

Previously Langdon and Petke [8] introduced a way to automatically generate the cube root function $\sqrt[3]{x}$ (cbrt) into the C Math library and automatically generate the lookup table required for it by using CMA-ES. We expand on their work, and show a way to generate a lookup table for any given mathematical function, within a predefined range. The goal is to provide a speedup for functions that need to be solved often during algorithmic evaluations and do not have hardware acceleration.

2   Background

2.1   Covariance Matrix Adaption - Evolution Strategy (CMA-ES)

The CMA-ES algorithm can be used to solve n-dimensional continuous numerical problems. It has been proven to work for local [4] and global [3] optimization. At the core of the algorithm is the covariance matrix of which a centroid is calculated that guides the search over several iterations of new population-generations by evolving a probability distribution. The essential operators in CMA-ES are mutation and crossover. Mutation happens around a standard deviation that is continuously updated during the run. Crossover is done by combining several individuals in the population to new points. Crossover and mutator are CMA-ES internal functions that are closely tied to the core covariance matrix, and were not adapted for our approach [4].

CMA-ES also does not require parameter tuning, as all values are calculated, and updated in regular intervals, internally around the core centroid [4]. Several parameters can be set, such as the initial standard deviation, an initial search position (centroid), but they only serve to speed up the algorithm by moving closer to an already known, or at least assumed, good global optimum, and an appropriate mutation size around it. Parameters relevant to the approach of this article will be discussed in the next section.

2.2   Evolving better Software Parameters

The publication of Langdon and Petke [8] discusses the need to automatically evolve software parameters, and emphasizes on applications in the domains of automated bug-fixing, maintenance of legacy code as applications in the field of Genetic Improvement [7]. As a proof of concept that software parameters can be improved automatically their publication shows how to generate the cube root (cbrt) function for the GNU C library (glibc) which does not implement it. The cbrt algorithm itself was not generated, but rather copied and modified from the IEEE square root implementation as is provided in glibc. CMA-ES was used to generate the lookup table that cbrt used with only three iterations of Newton-Raphson. The goal was to achieve IEEE 754 double precision accuracy (1 sign bit, 11 bit exponent, 52 bit fractional [5]).

Algorithmic Implementation of Cube Root - Langdon and Petke [8] adapted the existing cube root function of glibc, which does not perform a pure Newton-Raphson approximation, but does several refinements to extend the limited range of the lookup table (between 0.5 and 2) to the entire range of values double can take. This includes splitting the double value into two 32 bit components and performing bitwise operations on them. Three Newton-Raphson iterations are taken, and finally the values last bit is modified to ensure the closest possible rounding [10].

CMA-ES parameters - All parameters were left as default except the following. The problem size in [8] was of $N = 2$ as they selected values for both 32 bit components in cbrt. All stopping conditions in [8] were set to 0 to ensure the algorithm would only stop when it found the exact values required for the lookup table. The seed for the random number generation was set externally for reproducibility.

Restarting Strategies - CMA-ES can have several reasons why it fails to produce an exact result. The primary reason when generating lookup tables is that the fitness landscape becomes too flat in the area it is searching for, as all individuals in the population come close to perfect accuracy, but will not reach it, due to bitwise imprecision, or due to not randomly mutating to the final correct bit. [8] opted for a restart in this case with a different seed. In all cases of their function it was enough to run CMA-ES no more than 3 times to reach the closest possible answer.

Fitness function - When generating a lookup table for cube root every value in the lookup table represents a sub-range of the range the table was generated for. The fitness function in [8] used three test points, the lower end, the higher end and the middle of the range, for each table entry. The fitness function was evaluated by calling the cube-root with the spot in the lookup table initialized with the values in the evaluated individual of the CMA-ES population.

The fitness function did a logarithm conversion. All values except 0 had the absolute logarithm of *DBL_EPSILON* added to it. *DBL_EPSILON* in C is the minimal difference when added to 1 changes results in a different double value. All values below 1 had the logarithm taken as well. This essentially 'zooms' into the fitness when values extremely close to zero are dealt with.

Table 1: Analysis of Langdon and Petkes fitness function [8]. Modifying the fitness function with a *logarithm* has no effect. Their method is more accurate than the Java and C++ reference implementations. Total Error is the difference between x and $cbrt(x)^3$ over 512 test values.

Implementation	Distribution	Total Error $\times$ 10^{-10}
C - with log	even	3.1451
	random	3.3231
C - without log	even	3.1451
	random	3.3231
Java	even	3.3322
	random	3.6071
C++	even	6.2851
	random	7.2275

Listing 1.1: Conversion of qualities close to 0.

```
if (quality ==0.0) return quality;
if (quality < 1.0) return (−log(DBL_EPSILON))+log(quality);
return (−log(DBL_EPSILON))+quality;
```

2.3   Investigating Evolving better Software Parameters

As the CMA-ES stopping condition is targeted towards 0 already, and the standard deviation does reduce its size to a DBL_EPSILON during runs, this adaption to the fitness function should not impact the algorithm. To check this assumption we compared two different versions of Langdon and Petkes code. One of them was modified to not apply the logarithm in their fitness function. A batch file then applied this compilation process:

1. Compilation of the entire project, to ensure the CMA-ES algorithm runs no old versions.
2. Running the original Genetic Improvement script with a *seed* that the compilation script takes as input.
3. A script then created the new lookup table from the compilation results.
4. Recompilation of the project with the new lookup table.

A test harness generated values ranged between 0.5 and 10000. The amount of positions in the lookup table, 512 values, were evenly spaced inside the range (e.g. 0.5, 20, 30.5, ..., 9980.5, 10000). An additional 512 values were randomly selected inside the range. The test harness then randomly created 1000 seeds between 1 and 1000000. The compilation batch file was run with every seed, and all 1024 values were tested on that seed. The measurement was done by taking the result values given by the implementation and cubing them again. The difference between the original value and the re-cubed cube root values was calculated as the error. The total error is the sum of these over all test-values as

shown in Table 1. On all executed tests the results were equivalent, with every single seed, and in both versions of the code. This means that neither applying a log, nor selecting a seed has an impact in their approach.

Accuracy of the results - One noteworthy finding that is not mentioned in [8] is that their generated cube root function outperforms implementations of other programming languages as shown in Table 1 with Java and C++. We compared their algorithm not only to our adaption, but to the Java and C++ implementations of cbrt as well, and [8] outperforms all implementations.

3 Methods

We extended the original approach of [8], to be used for any function that can be approximated with the Newton-Raphson method. The method generates only the lookup table for a function defined by a developer. Our method can generate a lookup table with the parameters:

- Range - from a *lower end* to a *higher end*. The range restricts the space in the double values the lookup table is being generated for. This is necessary as not all functions can benefit from refinements such as the cube root.
- Table Size - The number of entries in the table essentially splits the range into sub-ranges. By increasing the size of the table precision in a smaller range can be improved. Alternatively a larger range can be covered with no loss of precision.
- User Function - the user function allows the user to define an entry point to handle operations in addition to the Newton-Raphson approach.
- Approximation function and its derivative - are required by the approach. They are used both in the fitness function of CMA-ES to generate the lookup table, and in the Newton-Raphson approach that uses the lookup table.
- Iterations - the number of iterations in the Newton-Rapson approach. Increasing the number of iterations can improve the range the lookup table can be used for, and improve upon the precision.

3.1 CMA-ES Settings

Listing 1.2: Cube Root implementation.

```
// Function for Newton−Raphson
double fn(const double approx) {
    return approx * approx * approx;
}

// Derivative of fn
double derivativeFn(const double approx) {
    return 3 * approx * approx;
}
```

Table 2: Analysis of the strategy to restart the algorithm if no exact value is found. Not restarting has a higher (better) mean. Restarting is not relevant.

	Mean	Std. deviation	Median	Min. exact values	Max. exact values
No Restarts	**496.33**	1.9646	496	491	**502**
3 Restarts	495.87	1.7271	496	**492**	499

```
// Function that allows user to modify input and result
double userFunction(const double x) {
    // accept negative numbers in cube root
    if (x < 0) return approximate(0.0 − x);
    if (x > 0) return approximate(x);
    return x;
}
```

Algorithmic Implementation - The algorithmic implementation was done with only Newton-Raphson. An example of the approximation function for cube root can be seen in listing 1.2.

CMA-ES parameters - Similar to [8] we did not change any of the default parameters of CMA-ES except the stopping conditions, which were set to 0 for the fitness as well. The seed is provided externally as well. Our method takes a problem size of 1 instead of 2, as the values will be selected for the entire double value instead of its 32 bit components.

Restarting Strategies - Langdon and Petke [8] opted to apply a restart in case the CMA-ES run did not find an exact value according to their fitness function. Their results showed that no more than 3 restarts were necessary and in most cases the first seed was acceptable. To check if this option impacts the results we compared 100 different runs without restarting, and 100 runs with restarting.

As Table 2 shows, the runs without any restarting have a higher mean and a higher maximum in the amount of exact numbers found. An analysis of the medians over 100 runs (same values used for no restarts / 3 restarts) showed that the differences are not statistically significant (expecting 5 out of 100 - p of 0.05). Shapiro Wilk shows (p 0.0023) which means the data is not normally distributed, Mann-Whitney-U for two independent samples shows a normalized (p 0.0566). Repeating the test multiple times with different sets of 100 runs showed similar behavior, sometimes even with statistical significance, with both no-restarts and 3 restarts having the better mean. This lets us assume that the restarts have less impact on the run than the random seed values. While restarts can positively impact the results due to choosing a new seed, omitting them greatly improves the runtime of the approach.

3.2   Test Setup and Measurements

To enable a better comparability over all tested root functions, as well as the different applied fitness functions, the range of the lookup table was set from 0.5

to 2. For all tests the table size was set to 512. All functions depend only upon the approximation and its derivative, with no additional steps taken to improve or change the results. To enable comparability with reference approaches the iterations of Newton-Raphson were fixed at 3. The tests in subsection 3.3 are conducted with 3 restarts, while the tests in section 4 were done with no restarts as the runs with inflection points proved to be too time-consuming.

The tests were always conducted over two separate sets of 512 values. One set was evenly spaced in the given range of 0.5 to 2, the second set was generated randomly using a uniform distribution. These two sets were always generated for one group of tests. The only thing changing when repeating the tests was the random seed value which was randomly selected between 1 and 1000000.

The tests show two different quality measurements:

- *Total Error* - which is calculated from applying the approximation function *fn* to the approximated value of a given input, and then subtracting that input from it. The *err*or value is always summed over all test-values to produce the total error.

$$\mathrm{err} = \mathrm{abs}(\mathrm{fn}(\mathrm{approximation}) - \mathrm{input})$$

- *Exact Values* - Which are the amount of values that were met exactly by Newton-Raphson using the fitness function. In the double range not all continuous numbers can be represented, so this measure takes into account if the approximation is the closest that could be represented with double. This is done by comparing the *err*or of the approximation, as well as one bit lower and one bit higher. The bit addition and substraction are conducted by copying the value into a long with memcpy adding or removing 1, and conducting another memcpy back to double.

$$\mathrm{exact} = \mathrm{err}(\mathrm{appr.}) \leq \mathrm{err}(\mathrm{appr.} - 1\mathrm{bit}) \&\& \mathrm{err}(\mathrm{appr.}) \leq \mathrm{err}(\mathrm{appr.} + 1\mathrm{bit})$$

3.3   Fitness function design

In subsection 2.2 we showed that applying a log to the fitness function had no impact. To check if this depends on the implementation of the algorithm we redid the test with our implementation of cube root as shown in listing 1.2.

The results (see Table 3) show that applying the logarithm not only has an effect, but that effect is statistically significant, with the logarithm application achieving better results. The fact that there is a deviation from the mean as well, means that the seed also seems to have an impact. We assume that this is due to the additional steps that the algorithm implements, which allows finding exact values with different initial seed values, making the algorithm more robust.

As applying a logarithm to CMA-ES does significantly impact the results we chose to compare several other methods of modifying the fitness function:

- *No mod.* - the fitness function without any modification.
- *Log.* - as was done in [8] adding log(quality) + log(DBL_EPSILON).

- *Inc. Log.* - it stands to reason that if the fitness function does benefit from applying a log that increasing the log value (== getting the value closer to zero) should provide more benefit. Thus, we applied log(quality) + log(DBL_EPSILON × DBL_EPSILON ×DBL_EPSILON) instead of just log(DBL_EPSILON).
- *Mul.* - Adding a logarithm has the benefit of representing smaller changes in the fitness function. A multiplication log(quality) ∗1000 should have the same effect.
- *Bitwise* - The actual double fitness value is copied into a long with memcpy, and then cast back to double. This sets the value equal to all bits that were off from zero. This modification brings the largest transformation, and ensures that all values that are just one bit off will result in a fitness of 1, while all exact values will have a fitness of 0.

In their original work Langdon and Petke decided on a fitness function that takes three values for every value in the lookup table. Those three values were the lower end, the upper end and the center of the range an entry in the lookup table represented (see a in Figure 1) [8]. The lookup table for Newton-Raphson does require a good staring point for all values covered in the range. Selecting both ends and the center of the sub-range, that one individual position represents, ensure a good starting position for the entire range.

There are other ways to represent the fitness function, and arguments to be made for each of them. We selected several options for comparison:

(a) *Outer* - The outer points - upper and lower end - and center of the range, which is the original from [8] (see a in Figure 1

(b) *Inner* - ⅓, the center and ⅔ of the range (see b in Figure 1). The argument for this option is that the points are more evenly distributed over the entire range, than a).

(c) *Center* - Only the center of the range (see c in Figure 1). To verify that there is cause in the assumptions of a) and b) that multiple points per lookup table entry make a difference.

Table 3: Analysis of the influence of applying *log* in our approach. *Log* improves the accuracy and makes a significant difference in most cases (bold).

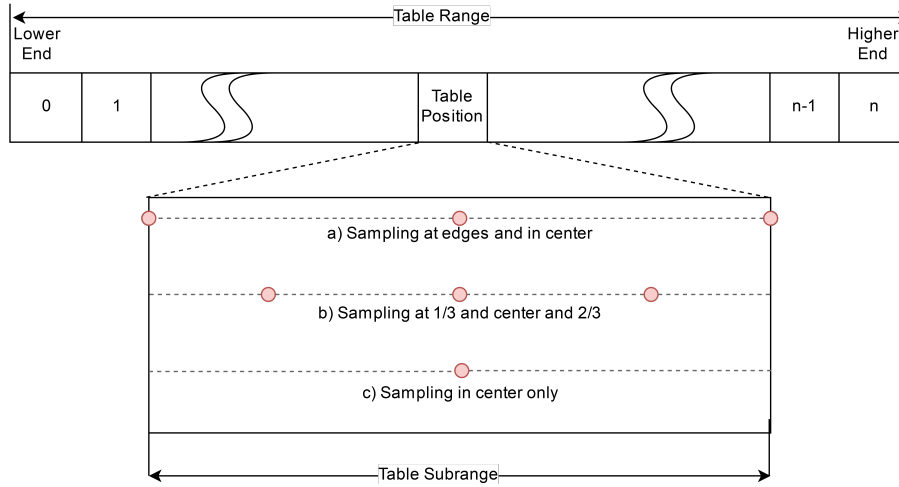| Distribution	Value	Fitness	Mean	Min	Max	Significance (p)
Even	Total Error × 10^{-14}	No log	9.02	8.93	9.13	1.5×10^{-133} (yes)
		Log	**8.62**	**8.49**	**8.73**	
	Exact Values	No log	472.64	467	478	3.7×10^{-124} (yes)
		Log	**493.06**	**487**	**501**	
Random	Total Error × 10^{-14}	No log	9.03	**8.78**	9.16	5.58×10^{-7} (yes)
		Log	**8.97**	8.8	**9.15**	
	Exact Values	No log	480.79	473	492	0.8929 (no)
		Log	480.86	471	489	

Fig. 1: Sampling point options (a-c) used in the fitness functions when generating a cell in the lookup table.

In these fitness functions there are several approaches as to what can be used to calculate the target for the lookup table positions. In all of them the goal is to set the currently searched lookup table position with the individual in the population to be evaluated and check that value for accuracy by applying it to the test-positions:

1. *Approx* - By comparing the result of the Newton-Raphson approximation, exactly as how the error in the tables is calculated
2. *Rem.Err* - By taking the last error after applying Newton-Raphson, essentially returning the difference instead of the desired result
3. *Direct* - By simply taking the individual of the population and applying the error function without applying Newton-Raphson at all. This is the most run-time efficient way to calculate a lookup table position as it requires only one call to fn(x) instead of three iterations, it is only viable when applying it to the center of the range.

When creating all valid combinations of the options above from the fitness function adaptions, test points and evaluation options a total of 35 functions have to be considered.

4   Results

To evaluate and validate our approach we selected these functions for testing:

– Square Root - as this has a reference implementation available in all languages (C, C++, Java)

- Cube Root - to offer a comparison with the work in [8]
- Super Root ($\sqrt[4]{x}$) - to provide new functionality in a similar area
- A polynomial with inflection point - to test how the lookup table behaves with an inflection point, which have difficulties for Newton-Raphson
- A function with many inflection points - to test what happens with multiple inflection points

Figure 2 shows the mathematical definitions of the above function and provides plots for them. For the polynomial with only one inflection point the lookup table range is outside of where the inflection occurs (around 0).

The results show that the approach is best suited for smooth functions, as the single inflection point influences the outcome. While most results can provide acceptable results (Error $\leq$ 1.5E-8) some runs fail to produce a valid lookup table. This happens even though the inflection is outside of the generated range for the lookup table. With multiple inflections inside the range not a single attempt generated an acceptable solution.

With the smooth square- cube- and super- root functions the fitness functions only taking the center point, and applying fn(x) instead of Newton-Raphson continuously provided good results in the random range. Using the outer test points and Newton-Raphson tended to produce better results in the even range. The fitness function continuously providing the worst results was using the center point and applying Newton-Raphson with the logarithm.

We attempted to test all algorithms with all fitness functions. This was achievable for square-, cube-, and super root. All results except the multiple inflection were calculated from 100 repeats. For the single inflection function we were only able to test 23 of the 35 defined fitness functions, as several took multiple hours per run to finish. For the multiple inflection function we were only able to test 15 repeats for all fitness functions.

The results for square root (see Table 4) cannot compete with the existing square root functions of all reference languages (C, C++ and Java). They do however show a trend that in the even distribution fitness functions that use the outer test values and approximate produce perfect results in the evenly distributed test set. This is similar over all functions without an inflection (see Table 5 and Table 6).

The results of Cube root (see Table 5) show the same trend as square root concerning the even distribution. Similar to the super root the fitness functions which only use the center point and apply the value of the approximation function directly instead of Newton-Raphson produce much better results in the randomly distributed set. Unlike the square root our approach is more accurate than C++ and Java in both test distributions.

The super root function behaves nearly exactly the same as the cube root concerning which fitness function works. The functions that used only the center test point and directly applied the approximation instead of Newton-Raphson had nearly the same results. This indicates that the smoother the function, the more consistent the approach becomes. The accuracy of the reference implemen-

(a) Square root $\sqrt{x}$
Approximation: x^2, Derivative: $2{\times}x$

(b) Cube root $\sqrt[3]{x}$
Approximation: x^3, Derivative: $3{\times}x^2$

(c) Super root $\sqrt[4]{x}$
Approximation: x^4, Derivative: $4{\times}x^3$

(d) Multiple Inflections $\sin(x{\times}40)$
Approx.: $\sin(40x)$, Deriv.: $40{\times}\cos(40x)$

(e) Polynomial with inflection $27x^3-3x+1$
Approx.: $27x^3 - 3x + 1$, Deriv.: $81x^2 - 3$

(f) Polynomial with inflection
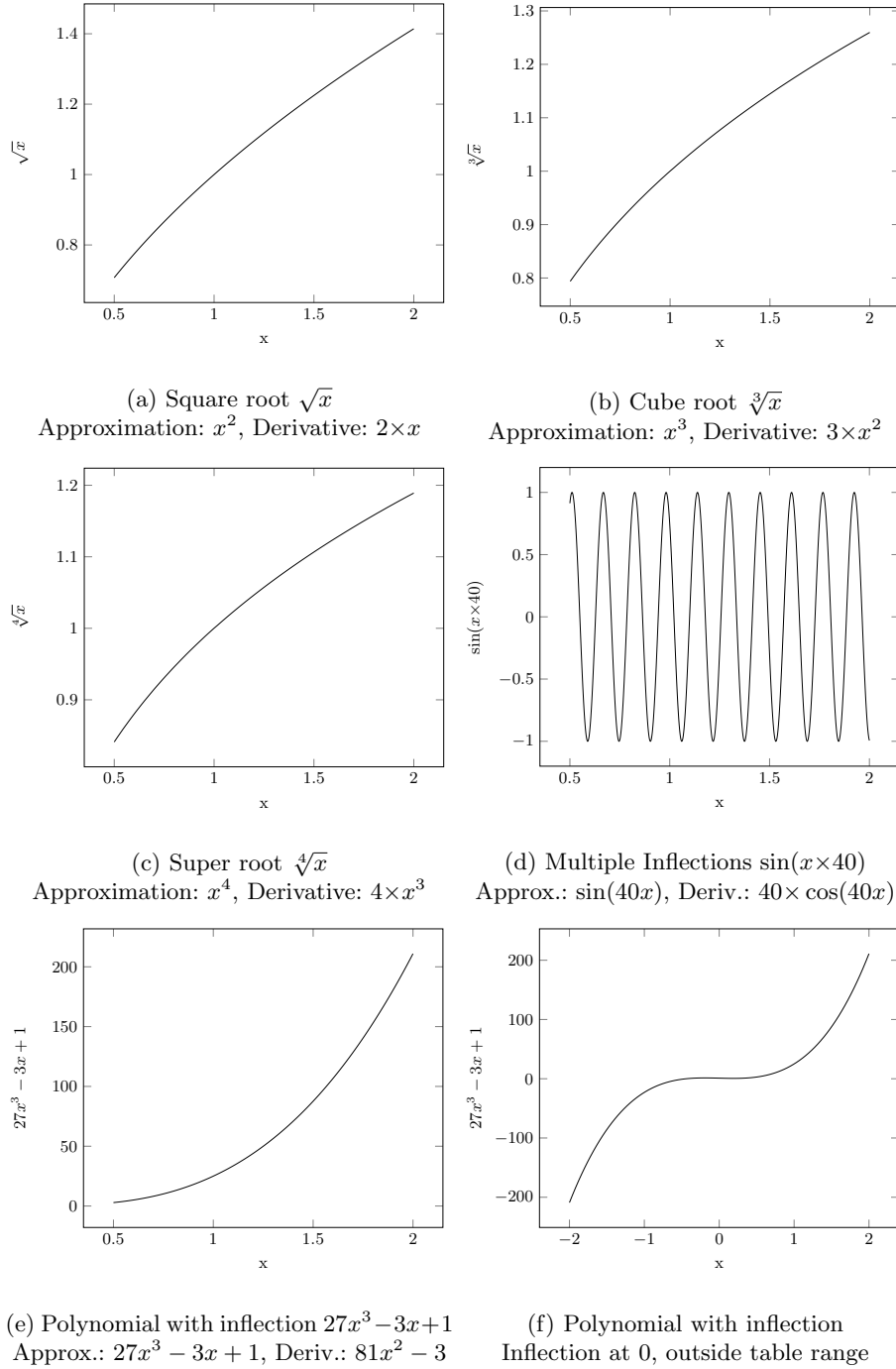Inflection at 0, outside table range

Fig. 2: Plots of functions under test in the 0.5 to 2 range the lookup tables were generated for. (f) shows an extended range of (e) which includes the inflection point around 0.

Table 4: Square root - comparison of lookup tables generated with different fitness functions. Our approach is less accurate than C, C++ and Java.

Distribution	Value	Fitness	Median	Min	Max
Even	Total Error $\times$ 10^{-14}	C Comparison	5.31	-	-
		C++ Comparison	5.31	-	-
		Java Comparison	5.31	-	-
		(Bitwise- Inc. Log.- Mul.- NoLog.-) Outer Approx	5.31	5.31	5.31
		Log Center Approx	6.58	5.55	12.5
	Exact Value	**(Bitwise- Inc. Log.- Mul.- NoLog.-) Outer Approx**	512	512	512
		Log Center Approx	465	284	500
Random	Total Error $\times$ 10^{-14}	C Comparison	5.27	-	-
		C++ Comparison	5.27	-	-
		Java Comparison	5.27	-	-
		(Bitwise- Inc. Log- Log- Mul.- No Mod.-) Center Direct	5.37	5.37	5.37
		Log Inner Approx	6.55	5.50	9.63
	Exact Value	**Log Center Rem. Err.**	506.5	499	511
		Log Center Approx	466.5	352	501

tation is not a valid comparison as we simulated the super root by applying the square root twice which results in a consequential error.

Table 7 shows that the approach can still work with a single inflection point. While the margin of error becomes considerable, several runs still managed to provide accurate results. Considering the amount of exactly calculated values it seems that points influenced by the inflection point can be problematic. A results table for the multi-inflection function is not provided as not a single run produced any value below a total error of 100 over 512 test values. Between 150-250 values still are calculated exactly, so this supports the assumption that points influenced by the inflection(s) are the source of the problem.

4.1   Run-Time Performance

The Run-Time performance of our approach is faster than Java, and slightly slower than the approach of Langdon and Petke. It is slower than approaches that are hardware accelerated. To enable a comparison we tested the cube root approximation of the approach against the C cube root of [8], and the native C++ and Java implementations. To have a baseline comparison for hardware accelerated functions in C we tested against the C square root as well.

Table 8 shows the run-time comparison from the total time taken when calling the respective functions 1,000,000 times. The benchmark was repeated 1000

Table 5: Cube root - comparison of lookup tables generated with different fitness functions. Our approach is more accurate than C++ and Java.

Distribution	Value	Fitness	Median	Min	Max
Even	Total Error $\times\ 10^{-14}$	Langdon and Petkes cbrt	8.33	-	-
		C++ Comparison	14.5	-	-
		Java Comparison	8.72	-	-
		(Bitwise- Inc. Log.- Mul.- NoLog.-) Outer Approx	8.33	8.33	8.33
		Log Center Approx	10	9.07	14
	Exact Value	**(Bitwise- Inc. Log.- Mul.- NoLog.-) Outer Approx**	512	512	512
		Log Center Approx	449	361	480
Random	Total Error $\times\ 10^{-14}$	Langdon and Petkes cbrt	8.78	-	-
		C++ Comparison	17.0	-	-
		Java Comparison	9.34	-	-
		(Bitwise- Inc. Log- Log- No Mod.-) Center Direct	9.13	9.1	9.17
		Log Center Approx	10.6	9.69	14
	Exact Value	**(Bitwise- Inc. Log- Log- No Mod.-) Center Direct**	492	491	493
		Log Center Approx	448	369	474

times, and the means over all tests is reported. For all executions the same argument was provided, and the function signature is the same, fn(double val).

Our approach is slightly slower than [8]. The difference is likely due to the number of multiplications, which is higher in our approach. It can not compete with the hardware accelerated functions of C and C++. All approaches for cube root perform better than Java. The approach was designed to enable generation of lookup tables for user provided functions, such as trust regions in Genetic Programming [14]. Hardware acceleration is not likely to exist for these cases.

4.2   Limitations

The greatest limitation of our approach is that it will not work on the entire range that the double data type can provide, but rather only for the range generated. As discussed in 2.2, reference implementations contain additional logic to ensure that algorithms like square root work over the entire double range. Our work concentrates on generating lookup tables for any given function, so these steps cannot be implemented since they would reduce the accuracy of the results when applied to a different function than intended.

Table 6: Super root - comparison of lookup tables generated with different fitness functions. Our approach is more accurate than C, C++ and Java, possibly due to the consequential error introduced by applying square root twice. This had to be done as the languages do not implement super root.

Distribution	Value	Fitness	Median	Min	Max
Even	Total Error $\times\ 10^{-13}$	C Comparison	1.32	-	-
		C++ Comparison	1.32	-	-
		Java Comparison	1.32	-	-
		(Bitwise- Inc. Log.- Mul.- NoLog.-) Outer Approx	1.18	1.18	1.18
		Log Center Approx	1.37	1.23	1.79
	Exact Value	**(Bitwise- Inc. Log.- Mul.- NoLog.-) Outer Approx**	512	512	512
		Log Center Approx	459	371	490
Random	Total Error $\times\ 10^{-13}$	C Comparison	1.30	-	-
		C++ Comparison	1.30	-	-
		Java Comparison	1.30	-	-
		Log Outer Rem. Error	1.19	1.17	1.21
		Log Center Approx	1.42	1.28	1.68
	Exact Value	**Log Outer Rem. Error**	490	482	496
		Log Center Approx	451.5	400	480

Table 7: Single Inflection function - comparison of lookup tables generated with different fitness functions. Our approach can still produce satisfying results though the inflection, which is outside of the lookup table range, has a negative impact on the achieved accuracy as shown by the large medians (10^{27}).

Distr.	Value	Fitness	Median	Min	Max
Even	Total Error	**Mul. Outer Rem. Error**	1.38×10^{-8}	7.14×10^{-14}	1.78
		(Mul.- No Mod.-) Center Direct	$2.2\ \times 10^{28}$	3.26×10^{25}	1.65×10^{33}
	Exact Value	**(Mul.- No Mod.-) Outer Appprox.**	510.5	488	512
		Log Center Approx	242	228	301
Random	Total Error	**Mul. Outer Rem. Error**	$1.9\ \times 10^{-11}$	7.51×10^{-14}	19.36
		(Mul.- No Mod.-) Center Direct	6.15×10^{27}	3.11×10^{24}	3.75×10^{31}
	Exact Value	**(Mul. Center- No Mod.-) Center Rem. Error**	435	410	451
		(Bitwise- Mul.- No Mod.-) Center Direct	239	229	295

Table 8: Run-time performance of root functions (average of 1,000,000 calls). Our approach is faster Java and nearly matches Langdon and Petkes approach, but can not compete with hardware acceleration.

	Mean (in nanoseconds per call)		
Language	sqrt	cbrt	surt
Hardware Accelerated C	0.88	0.88	0.88*
Hardware Accelerated C++	4.10	21.35	8.10*
Langdon and Petkes cbrt		24.96	
Our approach	25.33	27.46	29.58
Java	1.02	69.51	1.03*

*surt implemented as sqrt(sqrt(x))

The only currently known workarounds are increasing the size of the generated lookup table with the range allows keeping precision intact, while also increasing memory consumption. Alternatively increasing the allowed amount of Newton-Raphson iterations increases the range the lookup table can be generated for, at the cost of run-time performance.

The second limitation of the approach is that, due to Newton-Raphson, it cannot deal with functions that have inflections. While a single inflection point has a strong negative impact on result quality and the time it takes CMA-ES to generate the lookup table, a valid table can still be generated. With multiple inflection points generating a lookup table is not possible anymore.

5   Conclusions and Outlook

Generating lookup tables works well with smooth functions and can achieve double precision accuracy. Nearly all values can be approximated to the closest bit of a double. The run-time performance is in some instances faster than comparable software solutions, but can not compete with hardware accelerated functions.

That it is not able to equal [8] still shows that a well considered algorithm is more important than a good generation of constants with CMA-ES. The combination of robust algorithms with CMA-ES does provide the best results. In smooth functions however the approach consistently provides more accuracy than the reference implementation of C++ and to a lesser extent Java.

The results support the original findings of [8], that the application of genetic improvement techniques can be applied to create or update constants in programs. CMA-ES is especially a good fit as it manages its experiment parameters internally, and can deal with small (1×10^{-14}) differences in the search space. It is not robust against functions with inflection points. Even a single inflection in the function can hinder the approach.

The approach may also be applicable to any approximation function, such Gauss Newton, Aitken Extrapolation or Gradient Descent. Additionally, specializations to the resulting function (such as [10]) should be considered to reduce

the range limitation. In the future we also intend to use CMA-ES in Genetic Improvement as an operator to improve constant values in the population.

The source code, scripts and full results for tables 4-7 are available via `https://github.com/oliver-krauss/EuroGP2020-LookupTables`

References

1. Carli, R., Notarstefano, G., Schenato, L., Varagnolo, D.: Analysis of Newton-Raphson consensus for multi-agent convex optimization under asynchronous and lossy communications. In: 2015 54th IEEE Conference on Decision and Control (CDC) (Dec 2015). https://doi.org/10.1109/CDC.2015.7402236
2. Gordon, T.G.W.: Exploiting development to enhance the scalability of hardware evolution. Ph.D. thesis, University of London (2005)
3. Hansen, N.: Benchmarking a BI-population CMA-ES on the BBOB-2009 Function Testbed. In: GECCO 2009. ACM (2009). https://doi.org/10.1145/1570256.1570333
4. Hansen, N., Ostermeier, A.: Completely derandomized self-adaptation in evolution strategies. Evolutionary Computation **9**(2) (2001)
5. IEEE: Standard for Floating-Point Arithmetic. Std 754-2008 (Aug 2008). https://doi.org/10.1109/IEEESTD.2008.4610935
6. Koza, J.R.: Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems. Tech. rep. (1990)
7. Langdon, W.B.: Genetic Improvement of Software for Multiple Objectives. In: Search-Based Software Engineering. pp. 12–28. Springer (2015). https://doi.org/10.1007/978-3-319-22183-0_2
8. Langdon, W.B., Petke, J.: Evolving Better Software Parameters. In: Search-Based Software Engineering, pp. 363–369. Springer (2018). https://doi.org/10.1007/978-3-319-99241-9_22
9. Lenser, S.R., Tan, D.S.: Genetic Algorithms for Synthesizing Data Value Predictors. Tech. rep., Carnegie Mellon University (11 1999)
10. Markstein, P.W.: Computation of elementary functions on the IBM RISC System/6000 processor. IBM J. Res. Dev. **34**(1), 111–119 (Jan 1990). https://doi.org/10.1147/rd.341.0111
11. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes 3rd Edition: The Art of Scientific Computing. Cambridge University Press (Sep 2007), `http://dl.acm.org/citation.cfm?id=1403886`
12. Varagnolo, D., Zanella, F., Cenedese, A., Pillonetto, G., Schenato, L.: Newton-Raphson Consensus for Distributed Convex Optimization. IEEE Transactions on Automatic Control **61**(4) (Apr 2016). https://doi.org/10.1109/tac.2015.2449811
13. Yap, S.Z.Z., Zahari, S.M., Derasit, Z., Shariff, S.S.R.: An iterative Newton-Raphson (NR) method on Lee-Carter parameters estimation for predicting hospital admission rates. American Institute of Physics (AIP) Conference Proceedings **1974**(1) (2018). https://doi.org/10.1063/1.5041580
14. Z-Flores, E., Trujillo, L., Schütze, O., Legrand, P.: A Local Search Approach to Genetic Programming for Binary Classification. In: GECCO 2015. pp. 1151–1158. ACM (2015). https://doi.org/10.1145/2739480.2754797